SuperNOVA: a novel algorithm for graph automorphism calculations

RUSSELL K. STANDISH
Mathematics and Statistics
The University of New South Wales

The graph isomorphism problem is of practical importance, as well as being a theoretical curiosity in computational complexity theory in that it is not known whether it is NP-complete or P. However, for many graphs, the problem is tractable, and related to the problem of finding the automorphism group of the graph. Perhaps the most well known state-of-the art implementation for finding the automorphism group is Nauty. However, Nauty is particularly susceptible to poor performance on star configurations, where the spokes of the star are isomorphic with each other. In this work, I present an algorithm that explodes these star configurations, reducing the problem to a sequence of simpler automorphism group calculations.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on Discrete Structures*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph Algorithms*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: graph automorphism, Nauty, star-like graphs, computational complexity

1. INTRODUCTION

Given two graphs $g_1 = \{V, E_1\}$ and $g_2 = \{V, E_2\}$ where V is a set of labelled vertices, and $E_{\{1,2\}} \subset V \times V$ are sets of edges, the *graph isomorphism* problem is the problem of finding a permutation $\sigma : V \longrightarrow V$ of the vertices such that $\forall (i,j) \in E_1, (\sigma(i), \sigma(j)) \in E_2$. The map σ is known as an *isomorphism*. The graph isomorphism problem is of practical importance in applications such as storing and retrieving molecular structure data from a database[Kuramochi and Karypis 2006] or verification of printed circuit layout with respect to a schematic[Ebeling and Zajicek 1983]. It is also interesting, because it is not known whether the problem in general can be solved in polynomial time, or whether it is NP-complete.

A *graph automorphism* is an isomorphism of a graph onto itself. The set of graph automorphisms of a graph forms a group under composition. The graph automorphism problem is the problem of finding whether a graph has any automorphism other than the identity automorphism, which like the graph isomorphism problem has unknown computational complexity[Lubiw 1981]. More generally, one is interested in the size of the automorphism group[Standish 2005], and the orbits of the group.

The graph isomorphism problem can be reduced to the problem of finding a canonical labeling of the vertices of a graph. If the adjacency matrices of two graphs under their canonical labeling are equal, then they are isomorphic. For many graphs, finding a canonical labeling is tractable, and Nauty[McKay 1981] is probably considered one of the best-of-breed implementations. Nauty will also return the size of the automorphism group of a graph.

Unfortunately, Nauty struggles with star-like graphs, ie graphs where several isomorphic graphs are attached to each other via a single hub vertex. In this paper, I present the SuperNOVA, or "star exploder" algorithm, which can handle these sorts of graphs efficiently.

2. THE ALGORITHM

2.1 Canonical Ranges

The algorithm proceeds by defining an ordering relation on the graph vertices, sorting the vertices according to that ordering relation and the assigning a range of possible canonical labels to each vertex according to its position in the sorted list. For example, if the following sorted list was returned:

$$n_3 < n_2 = n_4 = n_5 < n_1 = n_0$$

then the vector of canonical ranges will look like

$$[4,5), [4,5), [1,4), [0,1), [1,4), [1,4).$$

(1) Initially, the ordering relation used is vertex degree (both in-degree and out-degree, and the number of bidirectional edges).

(2) Once all vertices have been assigned a canonical range, we can compare the canonical ranges of the nearest neighbours of a pair of vertices. If two vertices have the same canonical range, yet their neighbourhoods differ, we can further discriminate between the vertices, enabling a refinement of the canonical ranges. This step is repeated until no further refinement is possible.

The computational complexity lies between $O(n \log n)$ (the computational complexity of a sort) and $O(n^2 \log n)$ as at most n iterations can occur in step 2.

If the result of this algorithm is that every vertex has a canonical range of size 1, then we are done. The canonical labeling is given by the lower bounds of the canonical ranges, and there is only one automorphism (the identity). However, if some of the vertices have non-unit ranges, then the graph may have symmetries. Unfortunately, we cannot just take the product of the ranges as the size of the automorphism group, as not all such relabelings are automorphisms.

2.2 Symmetry Breaker

If the graph has symmetries, then at least two vertices will have identical canonical ranges. We need to determine which of the possible labelings is a canonical labeling. There may be more than one canonical labeling, but each such labeling produces an identical adjacency matrix. To compute a canonical labeling, we induce an ordering over adjacency matrices, and pick a labeling having the least adjacency matrix.

The outline of the symmetry breaker algorithm is:

```
If all canonical ranges are of size 1, then
    return an automorphism count of 1, and
        an adjacency matrix for that labeling,
otherwise
Find first non-unit canonical range [m, M)
    For each vertex j having canonical range [m, M),
        set vertex j's canonical range to [m, m + 1)
        apply step 2 of §2.1
        recursively apply the symmetry breaker algorithm to the new
            canonical ranges.
        add the returned automorphism count to the map entry indexed by
            the returned adjacency matrix
    return the least adjacency matrix and its automorphism count
```

The worst case scenario for this algorithm is when the sorting algorithm in §2.1 fails to discriminate vertices, in which case the complexity is $O(n!)$, as each permutation of vertices will be tried by the symmetry breaker. This will occur for the fully connected graph, which will always be a worst case, but also for the empty graph and star configurations. A star graph of order $n+1$ containing a single hub of degree n, and n leaf vertices, will cause the symmetry breaker algorithm to have complexity $O(n!)$. Given that this is the same problem that afflicts Nauty, this leads naturally to the star exploder algorithm.

2.3 Star Exploder

If we have a simple star topology, with c isomorphic graphs attached to a central hub, then the automorphism group size is given by $c!r$, where r is the automorphism group size of each of the spokes of the star, since there are $c!$ ways of relabeling the spokes. A slightly more general case occurs where there are c_0 spokes isomorphic to each other, another group of c_1 spokes isomorphic to each other and so on. In this case, the resulting automorphism group size is given by

$$r = \prod_i c_i! r_i. \tag{1}$$

To establish whether an arbitrary graph has a star-like topology, we remove all vertices with unit canonical range, which we call "fixed vertices". A graph colouring algorithm can be used to find the different contiguous subgraphs. If the graph breaks into more than one contiguous piece, then we can recursively apply the complete automorphism algorithm to each piece to obtain r_i, and count each piece using a map indexed by the canonical adjacency matrix. Then the overall automorphism group size can be found from the individual size by using equation (1). The overall canonical labeling can be found by using the algorithm described in §2.1, but with a modified ordering that includes information about which subgraph the vertices belong to (subgraphs sorted according to their canonical adjacency matrix order), and the canonical label of the vertex within the subgraph. If two vertices belong to different, but isomorphic subgraphs, and further that they have the same canonical label within their respective subgraph, then they are ordered

simply by their original label, as in this case it wouldn't matter which way they were labeled, the adjacency matrix would be identical. This allows a canonical labelling to be generated.

A subtle twist to be considered here is that a subgraph connected to one fixed vertex, and another subgraph connected to a different fixed vertex are not equivalent, even thought they may be isomorphic. To deal with this issue, we attach the vertex's canonical range from the original graph as an attribute to the equivalent vertex in the subgraph. Only isomorphic subgraphs whose attributes are identical are equivalent, otherwise they're counted as distinct graphs.

Star exploder fails when either there are no fixed vertices, or when removing all the fixed vertices does not partition the graph. Because the symmetry breaker algorithm gradually fixes more and more vertices at each level of recursion, the star exploder algorithm is applied at each level of recursion of the symmetry breaker algorithm, and will eventually succeed in breaking the graph into disjoint pieces. The worst case scenario is not so much the full graph (which being the dual of the empty graph is trivially transformed), but a digraph where each vertex is connected to every other vertex, and arranged so that the indegree and outdegree of each vertex is the same (the order of the graph must be odd for this to occur). Each vertex is the same as any other, so symmetry breaker must iterate over all $n!$ permutations of vertex labels.

3. IMPLEMENTATION AND RESULTS

The algorithm was implemented in C++ as part of the open-source *Ecolab*[Standish and Leow 2003] simulation environment, from ecolab.4.D31. onwards. It makes use of the C++ standard library sort() algorithm, and the standard associative containers map and set.

The algorithm was tested by comparing its calculated automorphism group size with that given by Nauty. If $S(g)$ is the canonical representation of g calculated by SuperNOVA, and $N(g)$ the canonical representation calculated by Nauty, a second important check is that $S(N(g)) = S(g)$ and that $N(S(g)) = N(g)$.

A database of 48940 symmetric graphs obtained from Brendan McKay's website (http://cs.anu.edu.au/~bdm/data/graphs.html) was used to check the equivalence of SuperNOVA with Nauty. Exhaustively generating all digraphs of a certain number of vertices and edges provided an independent test to ensure the algorithm worked for digraphs, including some with a star-like nature, however this was only feasible up to order 9 or so. Certain star-like digraphs extracted from the wiring diagram of a *C. elegans* brain was used to test the performance of SuperNOVA on graphs that proved intractable with Nauty. Attempting to run Nauty on these digraphs was unsuccessful, as Nauty didn't complete after several days of running, and had to be killed. By contrast, SuperNOVA computed these examples in seconds.

Figure 1 shows 1000 randomly generated Erdös-Rényi graphs with order $10 \leq n < 100$ and edge count $0 < l \leq n(n-1)/2$. Both SuperNOVA and Nauty were timed, and the times plotted as a function of order and edge count. Because some graphs can potentially take a very long to compute the canonical labeling, a maximum of 10 minutes was imposed on the computation by using CPU resource limit functionality

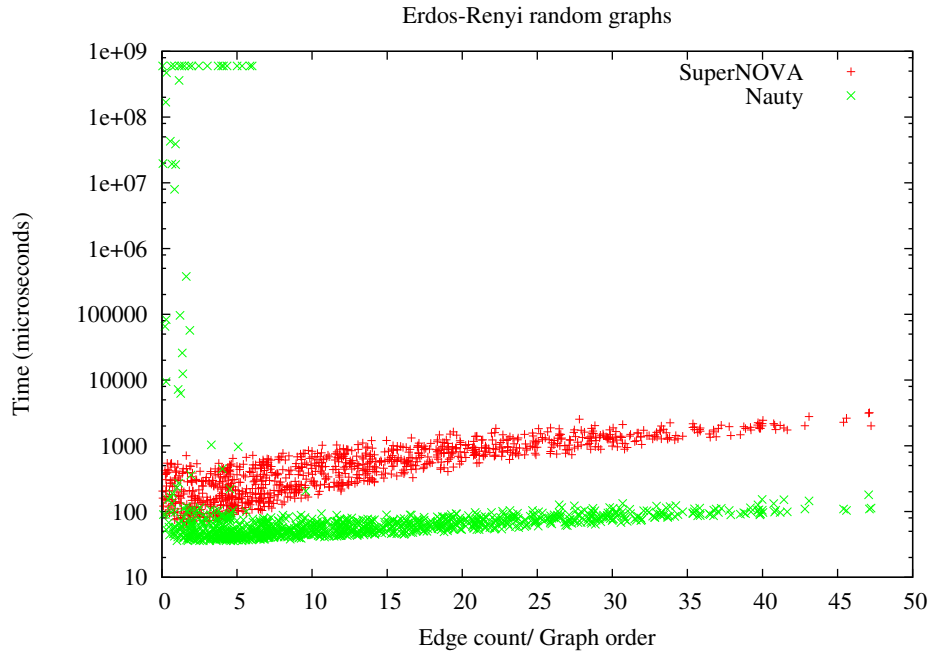

Fig. 1. Execution times for randomly generated Erdös-Rényi graphs with order $10 \leq n < 100$ and edge count $0 < l \leq n(n-1)/2$ by SuperNOVA and Nauty. Times are plotted against l/n. Whilst for most graphs, Nauty is an order of magnitude faster, for a number of graphs it is many, many times slower. These cases are all for low order/edge count ratios (< 7).

of Linux. These examples appear in the data as having an execution time of 10 minutes ($6 \times 10^8 \mu$s), and all from Nauty, and appear when $l/n < 10$.

Figure 2 shows the performance of SuperNOVA versus Nauty for the worst case scenario of a fully connected digraph, with the indegree and outdegree of each vertex being the same. As expected, SuperNOVA's execution time blows up very rapidly, but interestingly Nauty performs well, with polynomial time complexity.

4. DISCUSSION

SuperNOVA effectively handles the star-like configurations that give trouble to Nauty. On a sampling of 1000 Erdös-Rényi random graphs of order between 10 and 100, SuperNOVA computed the automorphism group size of all of the whole set within 10 minutes on a quad core Intel Core 2. By contrast, Nauty failed to complete the calculations on several of the graphs within the 10 minute time limit. Interestingly, all of these examples lie in the range $0 < l/n < 10$. Overall, Nauty is an order of magnitude faster than SuperNOVA on those examples it handles well, which is a reflection of the intense effort that has goine into the optimisation of that code. With more optimisation, SuperNOVA should be able to close the gap somewhat.

On the artificially constructed worst-case scenario, SuperNOVA performs poorly as expected, but Nauty performs well, executing in polynomial time.

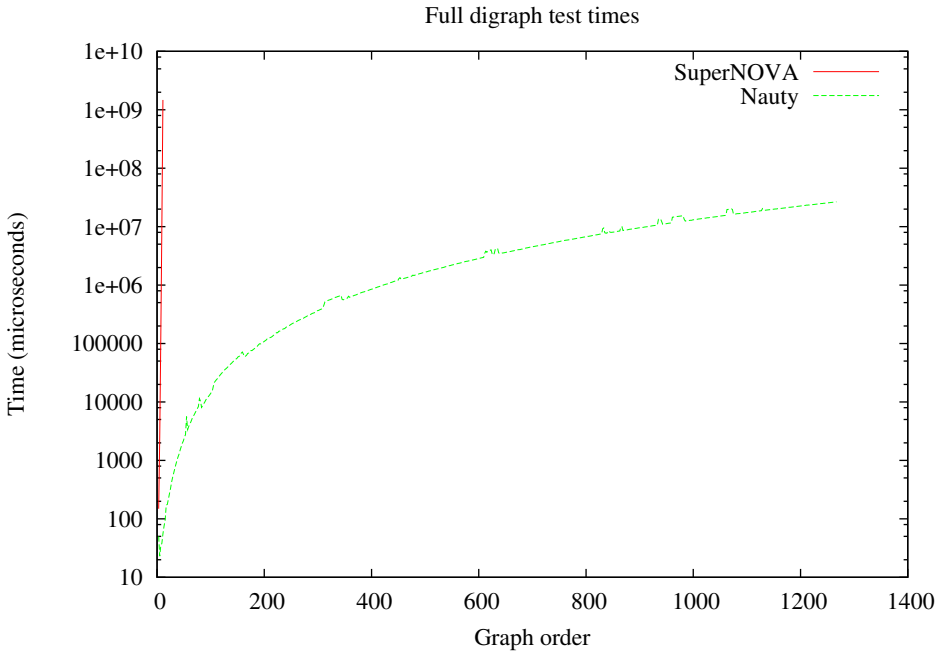

Fig. 2. Execution times for the full digraph case as a function of graph order. This is the worst case for SuperNOVA, which shows super exponential complexity. Nauty exhibits polynomial complexity

All of this suggests the possibility of a hybrid algorithm, leaving the sparse examples to SuperNOVA, and the denser examples to Nauty. The precise heuristic for determining which algorithm will need to be determined by future research. Furthermore, the precise canonical form returned differs in the two algorithms, so combining the algorithms will need to take this into account. For the isomorphism problem, it should not matter, so long as the same algorithm (SuperNOVA or Nauty) is applied to the two graphs being compared.

Finally, one may speculate as to whether the hybrid algorithm is truly polynomial complexity. Further work will be needed to try and identify worst case scenarios for both algorithms.

REFERENCES

Ebeling, C. and Zajicek, O. 1983. Validating VLSI circuit layout by wirelist comparison. In *Proceedings of the IEEE International Conference on Computer Aided Design (ICCAD-83).* 172–173.

Kuramochi, M. and Karypis, G. 2006. Finding topological frequent patterns from graph datasets. In *Mining Graph Data*, D. J. Cook and L. B. Holder, Eds. John Wiley and Sons, 117–158.

Lubiw, A. 1981. Some NP-complete problems similar to graph isomorphism. *SIAM Journal on Computing 10,* 1, 11–21.

McKay, B. D. 1981. Practical graph isomorphism. *Congressus Numerantium 30*, 45–87.

Standish, R. K. 2005. Complexity of networks. In *Recent Advances in Artificial Life*, Abbass

et al., Eds. Advances in Natural Computation, vol. 3. World Scientific, Singapore, 253–263. arXiv:cs.IT/0508075.

Standish, R. K. and Leow, R. 2003. EcoLab: Agent based modeling for C++ programmers. In *Proceedings SwarmFest 2003*. arXiv:cs.MA/0401026.